Assignment 4 Report

By Aniruddha Bala
Roll No: 15655

December 1, 2018

1 Parallelization of KMeans Clustering using OpenMP

1.1 Methodology

1.1.1 Visualizing the data

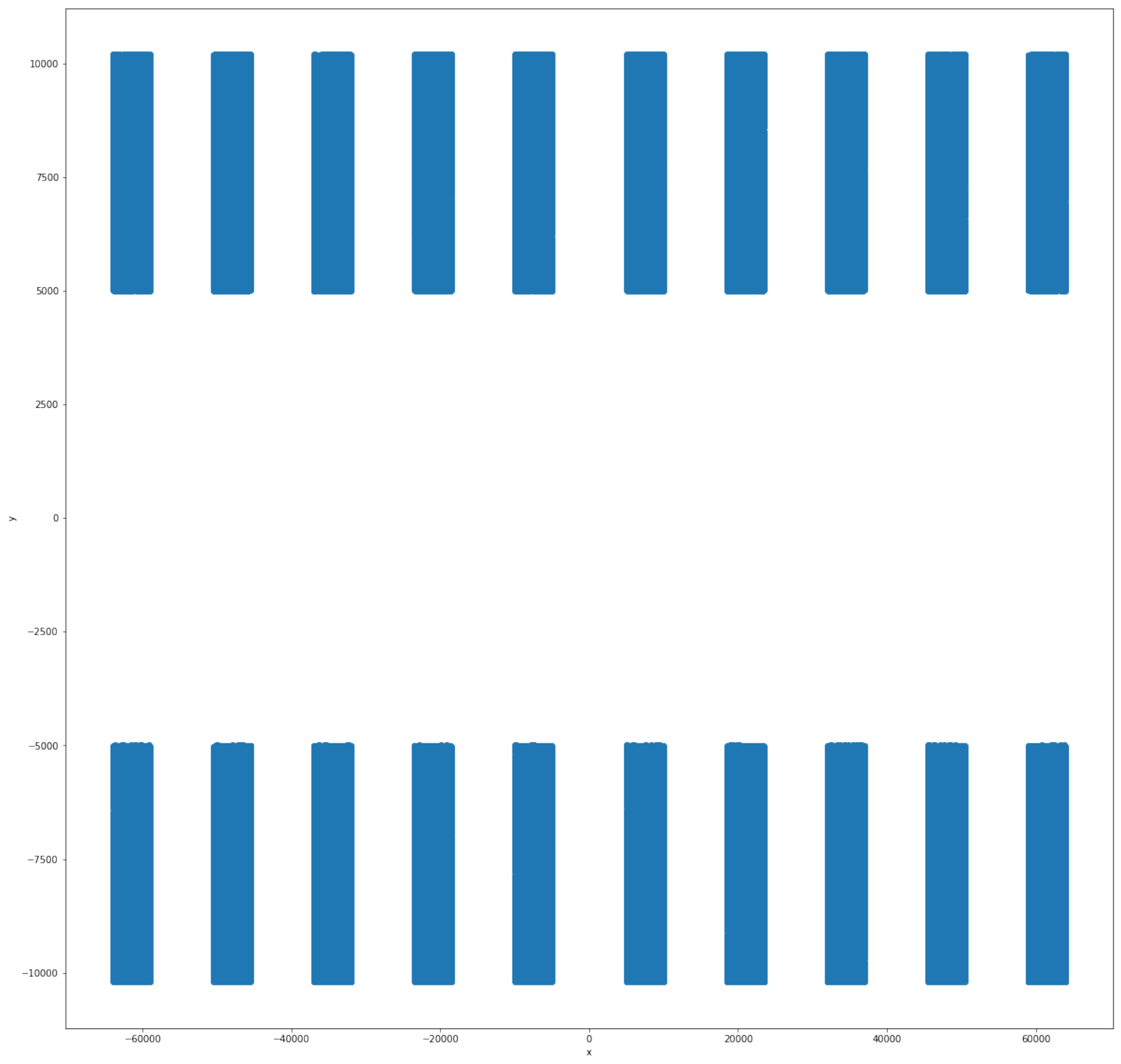
At first the data was read from input file and the data points were plotted in python using matplotlib. The data was found to have a good natural structure arranged in 20 blocks. The plot obtained is shown below.

```
In [2]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
```

```
In [98]: data = pd.read_csv('../openmp/data_500k.csv', header=None)
```

```
In [99]: plt.figure(figsize = (20,20))
         plt.scatter(data[0],data[1])
         plt.xlabel('x')
         plt.ylabel('y')
```

```
Out[99]: Text(0,0.5,'y')
```



1.1.2 Sequential Kmeans

Then I implemented the sequential program for KMeans Algorithm. The algorithm takes n points as input and forms K clusters out of it. In this case we had 500k points and we had to group them into 20 clusters. The KMeans algorithm works as follows:

1. Initialize random centroids for each of the K clusters
2. Repeat until convergence do:
3. Cluster assignment step: For each of the n points assign them to a cluster(i) whose centroid is closest to the point.
4. Move centroid step: For each of the clusters update the cluster centroid to the average of points assigned to that cluster.

1.1.3 Parallelizing with OpenMP

In each loop the major computation in the Kmeans algorithm is done in the part where we assign cluster centroids to each of the n points. Therefore parallelizing that loop makes sense. The loop has been parallelized using the openmp for directive. The parallelization has been tried for 4, 8, and 16 number of threads and using static and dynamic scheduling for assigning iterations to threads.

1.2 Experimental setup

Input data: The input data file data_500k.csv contains the (x,y) coordinates of 500000 points.

Output : The program outputs 20 lines each line containing the cluster index of the cluster formed, the number of points in the cluster, centroid coordinates of the cluster. At the end the execution time of the program is also printed.

The following steps were followed while conducting the experiment:

1. Run the sequential program, note the result and the execution time.
2. Run the parallelized openmp version with all possible combinations of number of threads (4, 8, 16) and schedule clause (dynamic and static)
3. Note the execution time in each case.
4. To maintain uniformity both the sequential and parallelized versions of the code have been timed with openmp's high resolution timer function omp_get_wtime().

1.3 Results

1.3.1 Output

Result below shows the 20 cluster centroids to which the algorithm converged

```
In [100]: kmeans_op = pd.read_csv('../openmp/result.csv')
          kmeans_op.head(20)
```

```
Out[100]:     CLUSTER_ID  NUM_PTS   CENTROID_X   CENTROID_Y
          0            0    25000   7507.21508  7591.16904
          1            1    25000  20986.22856  7583.28724
          2            2    25000  34505.74344  7590.43368
          3            3    25000  47995.84272  7603.73724
          4            4    25000  61505.01472  7615.65620
          5            5    25000  -7509.70652  7600.08716
          6            6    25000 -20999.28700  7604.38880
          7            7    25000 -34503.96696  7606.82912
          8            8    25000 -47997.90336  7590.85656
          9            9    25000 -61505.04916  7603.93612
          10          10    25000  -7496.48468 -7599.07976
          11          11    25000 -21017.49684 -7574.37808
          12          12    25000 -34508.74520 -7605.09100
          13          13    25000 -47999.49952 -7606.13836
```

```
14          14    25000 -61518.25144 -7600.48044
15          15    25000   7505.19200 -7591.39480
16          16    25000  21011.26088 -7583.93400
17          17    25000  34495.05208 -7616.55264
18          18    25000  48001.69036 -7579.16096
19          19    25000  61501.02676 -7603.22488
```

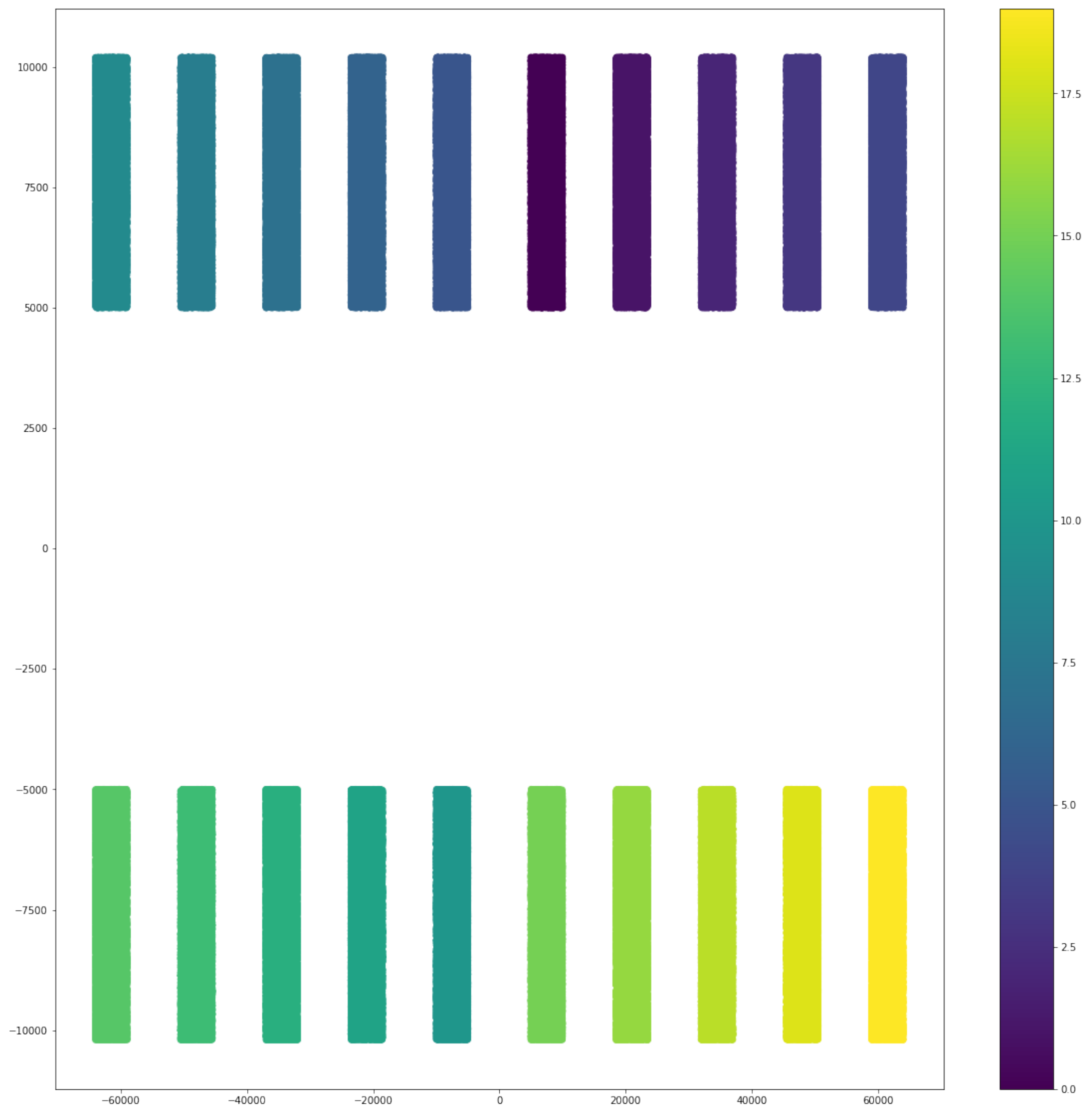
Figure below shows the different clusters formed in different clusters after the program is run.

```
In [101]: clusters_formed = pd.read_csv("../openmp/clusters_formed.csv", header = None)
```

1.3.2 Visualization of the clusters formed

```
In [21]: plt.figure(figsize = (20,20))
         plt.scatter(data[0], data[1], c = clusters_formed[0])
         plt.colorbar()
```

```
Out[21]: <matplotlib.colorbar.Colorbar at 0x7f3da4618710>
```



1.3.3 Execution times in msecs

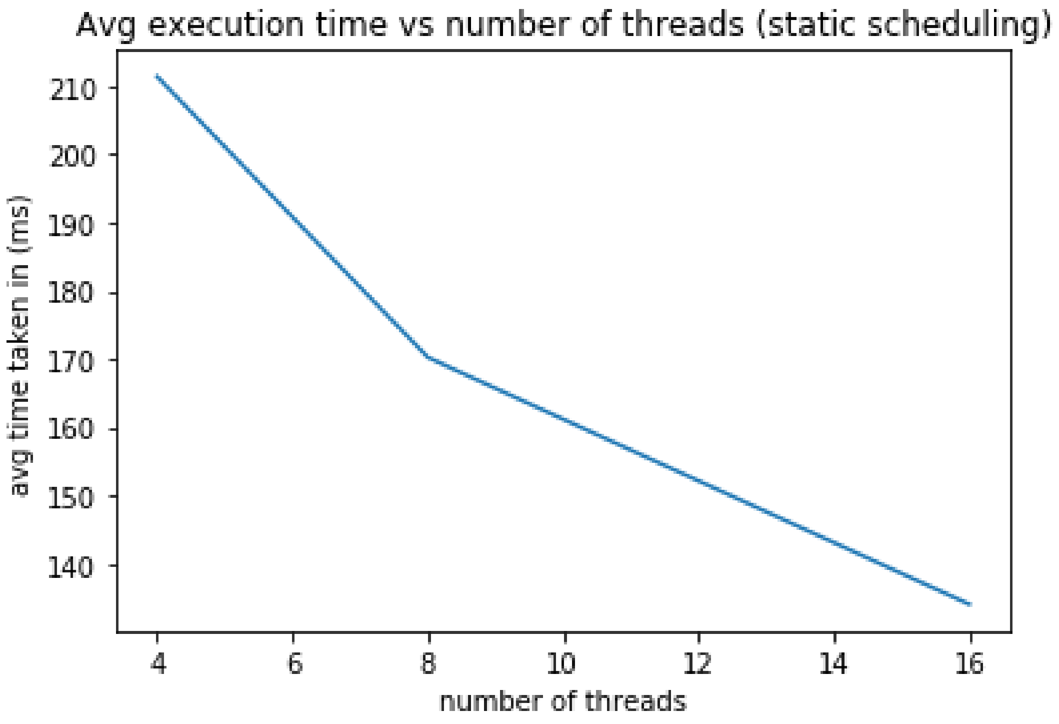
```
In [3]: openmp_timings = pd.read_csv('../openmp/openmp_timings.csv')
        openmp_timings.head()
```

```
Out[3]:       seq  static 4  dynamic 4  static 8  dynamic 8  static 16  dynamic 16
        0  537.630   191.073    292.110   175.893    249.565    135.812     248.685
        1  527.383   184.151    250.661   196.138    253.082    123.065     247.491
        2  568.125   193.157    244.271   171.515    291.176    131.803     248.790
        3  528.254   263.305    291.080   175.525    253.013    130.494     246.350
        4  564.453   225.322    294.039   132.152    252.753    149.175     250.485
```

1.3.4 Plots

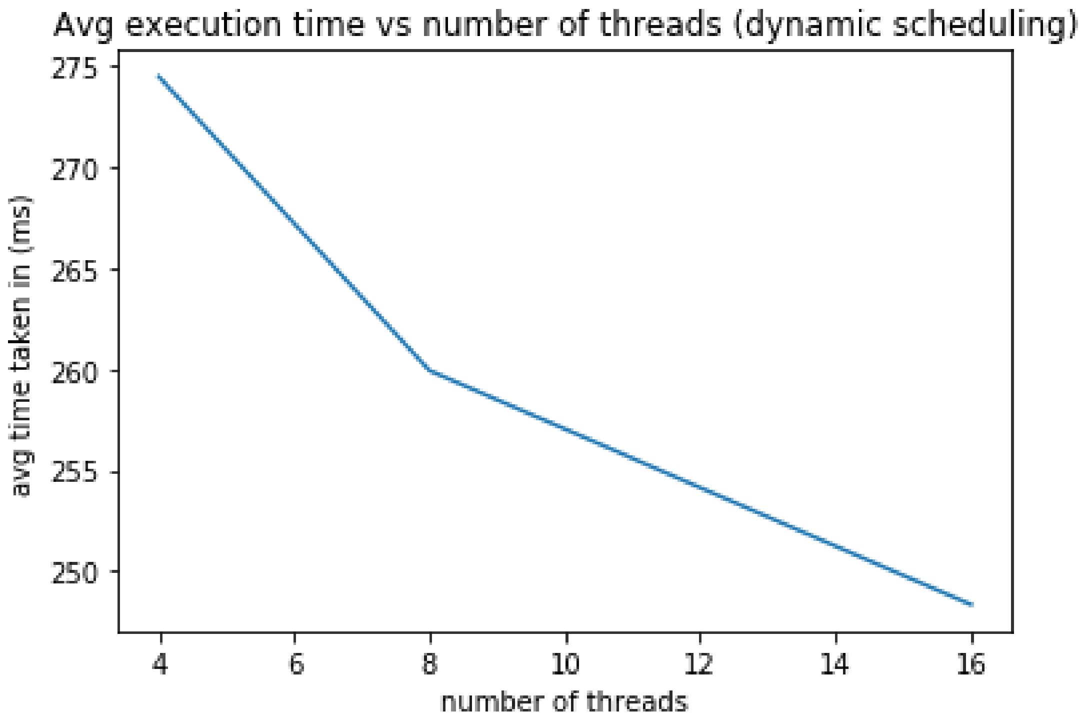
```
In [4]: openmp_avg_times = openmp_timings.mean().values
        seq_time = openmp_avg_times[0]
        openmp_avg_times = openmp_avg_times[1:]
        dynamic_idx = [1,3,5]
        static_idx = [0,2,4]
        nthreads = [4,8,16]
        plt.plot(nthreads, openmp_avg_times[static_idx])
        plt.xlabel('number of threads')
        plt.ylabel('avg time taken in (ms)')
        plt.title('Avg execution time vs number of threads (static scheduling)')
```

```
Out[4]: Text(0.5,1,'Avg execution time vs number of threads (static scheduling)')
```

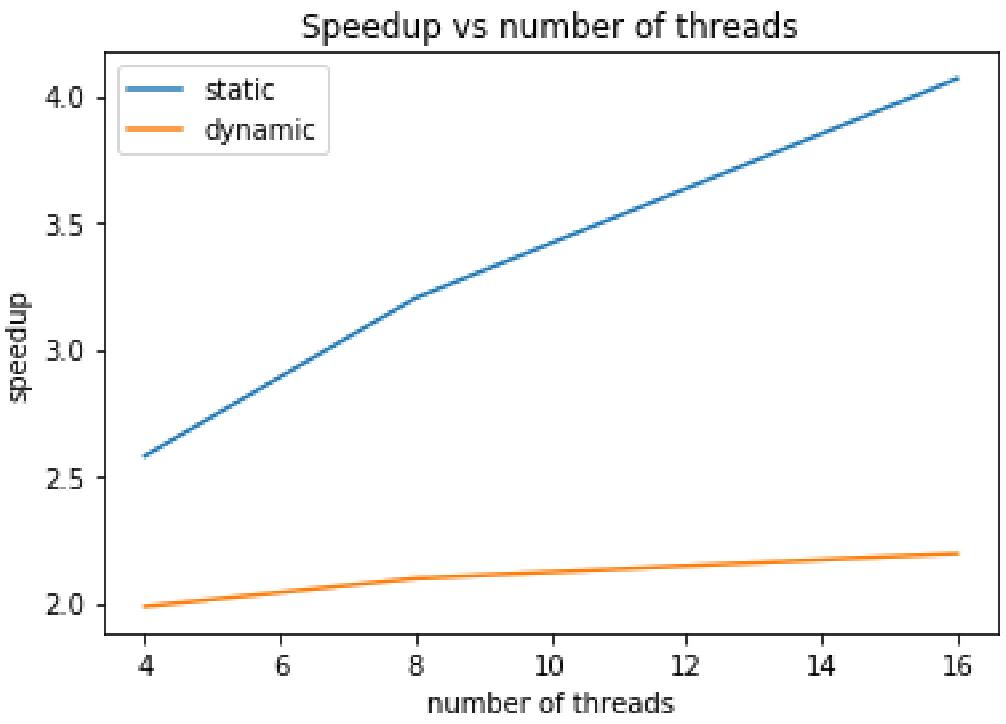


```
In [5]: plt.plot(nthreads, openmp_avg_times[dynamic_idx])
        plt.xlabel('number of threads')
        plt.ylabel('avg time taken in (ms)')
        plt.title('Avg execution time vs number of threads (dynamic scheduling)')
```

```
Out[5]: Text(0.5,1,'Avg execution time vs number of threads (dynamic scheduling)')
```



```
In [6]: speedup_static = [seq_time/openmp_avg_times[i] for i in static_idx]
        speedup_dynamic = [seq_time/openmp_avg_times[i] for i in dynamic_idx]

        plt.figure()
        plt.plot(nthreads, speedup_static)
        plt.plot(nthreads, speedup_dynamic)
        plt.xlabel('number of threads')
        plt.ylabel('speedup')
        plt.title('Speedup vs number of threads')
        plt.legend(['static','dynamic'])

Out[6]: <matplotlib.legend.Legend at 0x7f3ba7127f28>
```



1.4 Observations

1. Due to well structured data with proper choice of initial centroids we obtain the desired clusters in less number of iterations.

2. Every iteration of sequential Kmeans is roughly $O(npoints * nclusters)$, and overall running sequential Kmeans on 500K points takes on an avg 520 msec.

3. Parallelizing the loop with openmp can give substantial speedups for eg. for 4 threads we obtain almost 2.5x speedup

4. With increasing number of threads we obtain higher speedups however after a certain point we can expect to hit a saturation point after which the curve saturates. This is in accordance to Amdahl's law.

5. The speedups obtained through dynamic scheduling are lesser as compared to static, this is because the workload(parallelized loop) is regular among the threads hence we dont see much performance boost.

2 MPI Merge Sort

2.1 Methodology

In this we implement a parallel merge sort program in mpi that can sort a list of integers. The idea here is process 0 first gets the data in global array this data is then scattered to the individual processes local array using mpi function MPI_Scatterv(), the reason I have used MPI_Scatterv is because here the data may not be divided equally among the processors in case the number of integers in the array is not divisible by the number of processes. In such a case each processor first gets N/nprocs number of integers and any leftover integers are assigned equally among the processors instead of assigning all remaining ints to one processor. Then the parallel_merge_sort() function is called for each of the threads. The parallel merge sort works as follows:

1. Each of the processes first sort there part of the local array using stl function quick sort.
2. After that a loop runs for log2(nprocs) number of levels
3. Each of the processor finds if it is the root/parent processor if it is not then it sends its data (local array and size) to the root processor using an MPI_Send.
4. The root processor receives the data from its partner using MPI_Recv and merges the two arrays and updates its array to the merged array and also updates its size.
5. Finally the result is obtained in processor 0.

2.2 Experimental Setup

Input: The input data file merge_data.txt contains 1L integers. Output: The output of the algorithm is sorted list of integers along with the execution time for running the program The output is generated in result.csv file

The experiment was conducted for 8, 16, 32, 64 and 80 processes and timings for each were noted and compared with sequential quick sort. The total execution time for the algorithm is obtained by calculating the maximum time across all the processes.

2.3 Results

2.3.1 Output

For the sorted output please refer the result.csv file

2.3.2 Execution times in msec

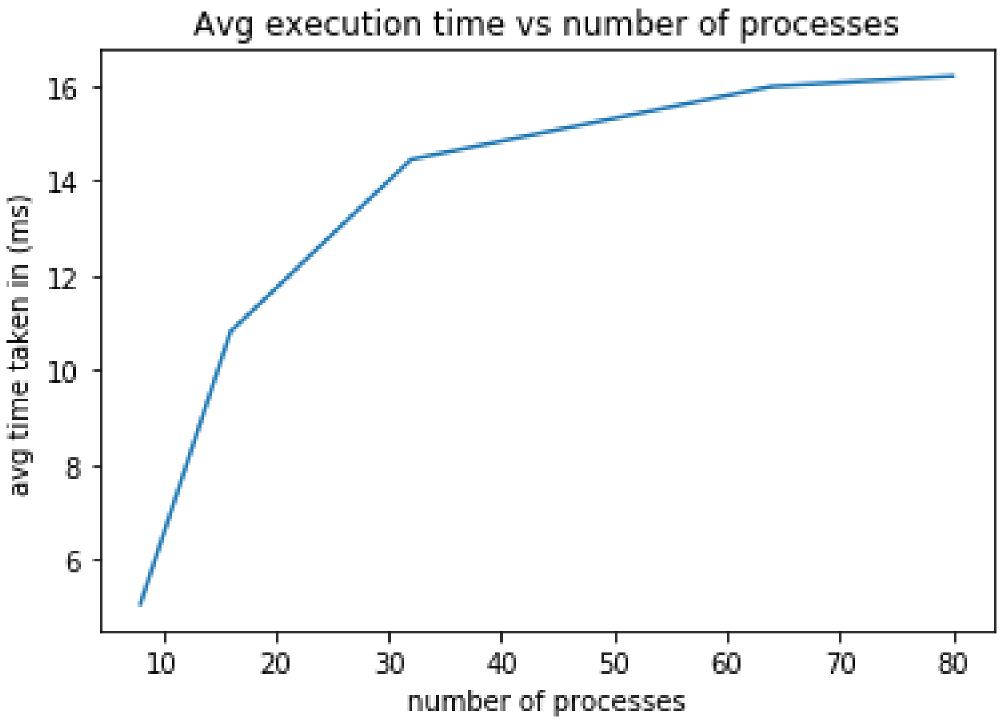
```
In [7]: sort_times = pd.read_csv('../mpi/mpi_result.csv')
        sort_times.head()
```

```
Out[7]:    seq sort   8 procs   16 procs   32 procs   64 procs   80 procs
        0    15.173  5.115271   8.249760  13.636112  27.039051   6.471157
        1    15.606  5.060673   8.991241   9.444475  11.289597  16.919613
        2    15.156  5.105257   8.766890  14.188051  11.272907  15.195131
        3    15.062  5.049944  13.677359  15.269756  18.224955  14.958858
        4    14.891  5.011320  14.421225  19.699812  12.118101  27.476549
```

2.3.3 Plots

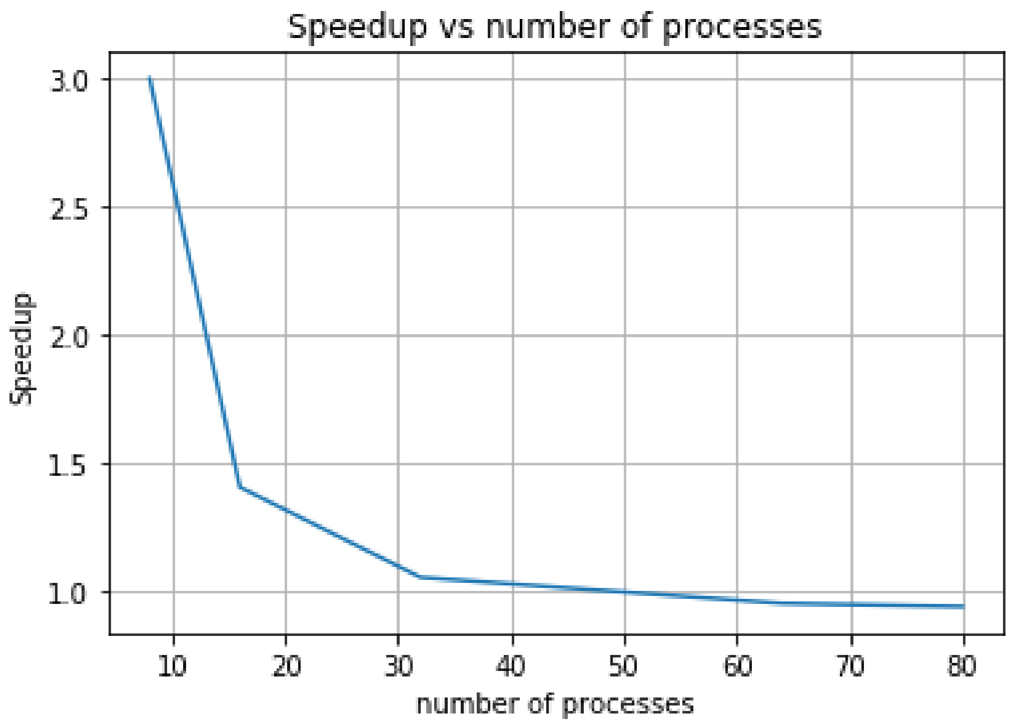
```
In [8]: mpi_avg_times = sort_times.mean().values
        seq_time = mpi_avg_times[0]
        mpi_avg_times = mpi_avg_times[1:]
        nprocs = [8,16,32,64,80]
        plt.plot(nprocs, mpi_avg_times)
        plt.xlabel('number of processes')
        plt.ylabel('avg time taken in (ms)')
        plt.title('Avg execution time vs number of processes')
```

```
Out[8]: Text(0.5,1,'Avg execution time vs number of processes')
```



```
In [9]: speedup = [seq_time/i for i in mpi_avg_times]
        plt.plot(nprocs, speedup)
        plt.xlabel('number of processes')
        plt.ylabel('Speedup')
        plt.grid()
        plt.title('Speedup vs number of processes')
```

```
Out[9]: Text(0.5,1,'Speedup vs number of processes')
```



2.4 Observations

1. C library quicksort is highly optimized and has a linearithmic running time and takes on an average 15 msecs to run.
2. For the given 100K dataset maximum speedup is obtained with 8 processes, with a speedup factor of almost 3.
3. With increasing number of processes the communication overhead overshadows any performance benefits that we may expect to see, therefore the speedup curve decreases with increasing number of processes.

3 Reduction using CUDA

3.1 Methodology

Here we implement tree based reduction of a array of integers using CUDA. Two different versions of reduction is implemented one using global memory and one using shared memory. As here we are working with 20K ints and using only one thread block and 1K threads, each thread first reduces 20 elements each, after all the threads are done we are left with 1K ints on which we apply the conventional tree based reduction algorithm to get the result. The program goes as follows:

1. Allocate memory and copy data to device memory.
2. Launch the kernel with 1 block and 1K threads and arguments as d_data(device data array), res (result array), nprocs (number of processes).

3. Each thread reduces 20 elems each at the end of which we synchronize.
4. The we start with N/2 threads and reduce two elements at a time until we are left with a single thread.
5. Finally the result is stored in the res[0].
6. Copy back the result from device to host.

3.2 Experimental Setup

Input: The input data file reduce_data.txt contains 20K ints. Output: The output of the program is a single integer containing the reduction result along with the execution time.

The experiment was conducted twice once for the res array allocated in global memory and once for the res array allocated in shared memory. The execution times obtained in two cases were recorded and compared.

3.3 Result

3.3.1 Output

Cuda Reduction result: 637144 Actual answer: 637144

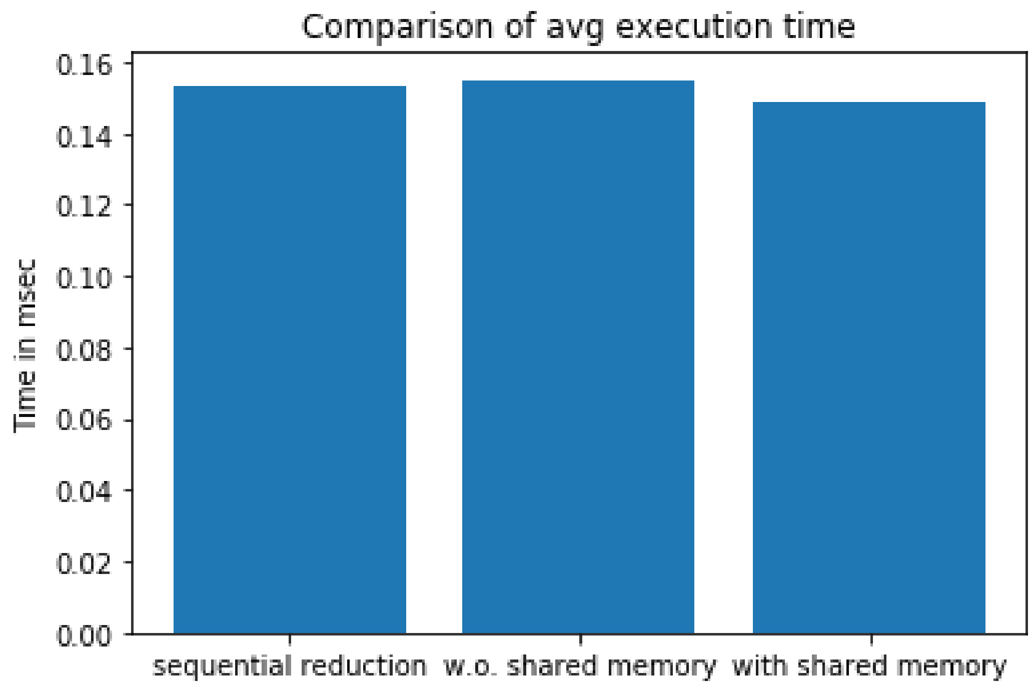
3.3.2 Execution times in msec

```
In [112]: reduction_times = pd.read_csv('../cuda/cuda_results_latest.csv')
          reduction_times.head() #time in msecs

Out[112]:         seq  w.o shared    shared
          0  0.164458    0.156192  0.146400
          1  0.150597    0.153728  0.147264
          2  0.150415    0.158784  0.151904
          3  0.150508    0.154688  0.150112
          4  0.150488    0.151392  0.148224
```

```
In [113]: reduction_times.mean()

Out[113]: seq           0.153293
          w.o shared    0.154957
          shared        0.148781
          dtype: float64
```

3.3.3 Plots

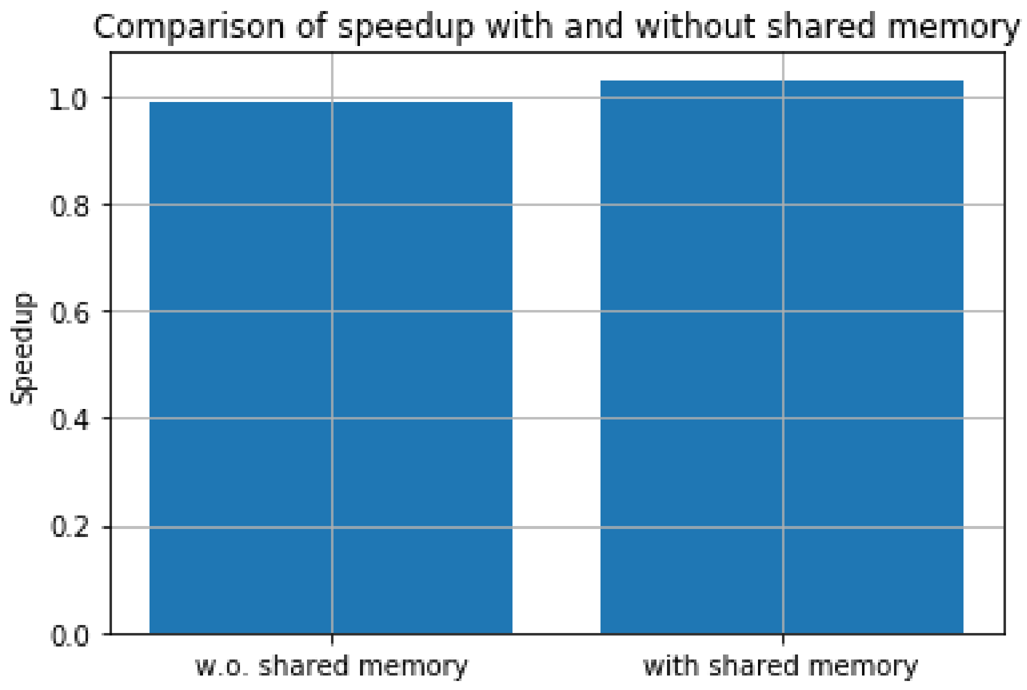
```
In [114]: y = reduction_times.mean().values
          x = ['sequential reduction','w.o. shared memory','with shared memory']
          plt.bar(x,y)
          plt.ylabel('Time in msec')
          plt.title('Comparison of avg execution time')

Out[114]: Text(0.5,1,'Comparison of avg execution time')
```



```
In [115]: speedup = [y[0]/i for i in y[1:]]
          print(speedup)
          x = ['w.o. shared memory','with shared memory']
          plt.bar(x,speedup)
          plt.ylabel('Speedup')
          plt.grid()
          plt.title('Comparison of speedup with and without shared memory')
```

```
[0.9892641045762435, 1.0303291822600769]
```

```
Out[115]: Text(0.5,1,'Comparison of speedup with and without shared memory')
```



3.4 Observation

1. Since the dataset size is only 20K and parallelization has been done with 1000 threads in 1 block the performance of sequential reduction and parallel reduction using cuda is almost similar.
2. We can further optimize the performance of CUDA program by using shared memory, using shared memory the speedup improves from 0.9892 to 1.03